A Demonstrably Correct Compiler

Susan Stepney, Dave Whitley, David Cooper and Colin Grant
Logica Cambridge Ltd, Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK

Keywords: Correct compiler; Formal specification; Denotational semantics; Prolog; DCTG

Abstract. As critical applications grow in size and complexity, high level languages, rather than better-trusted assembly languages, will be used in their development. This adds potential for extra errors to creep in, especially in the now necessary compiler. To avoid these new errors, it is necessary to have a formal specification of the high level language, and a formal development of its compiler. We outline what we believe is a practical route for achieving a demonstrably correct compiler, and describe a prototype compiler we have built by this route for a small, but non-trivial, language.

1. Introduction

It has been argued that the only "safe" way to write critical applications is by using assembly language, because this is the only way to be sure about what will happen during program execution. Languages further removed from the hardware cannot be trusted for two reasons:

- It is impossible to know what the language means; to know how it translates to the "real" machine.
- Even given some idea of the meaning of the high level constructs, it is impossible to know that the compiler correctly implements this meaning.

There is a grain of truth in this argument, but as the applications grow larger and more complex, using assembly language becomes infeasible; high level languages, with all their software engineering advantages, will become essential. How can these conflicting requirements be reconciled? As a first step along the way, (at least) the following need to be satisfied:

Correspondence and offprint requests to: Susan Stepney, Logica Cambridge Ltd, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, UK.
• It must be possible to deduce the logical behaviour of a particular program independent of its execution on a particular target.
• The high level language must have a formally defined semantics. Otherwise it is impossible to deduce even what should be the effect of executing a particular program.
• The formal semantics must be established and made available for peer review and criticism.
• The compiler must be correct, hence it must be derived directly from the formal semantics.
• The compiler for a critical language must be seen to be correct. Hence it must be written legibly, and must be easily related to the formal semantics.
• The code produced by the compiler must be clear, and easily related to the source code. This gives the required visibility to the compilation process for a critical language.

The last two points are important for critical applications, in order to conform with the much more stringent validation requirements these have.

In addition to the above requirements, the equally thorny problems of showing that the application is correct, and of showing that the hardware correctly implements the meaning of the machine language, must be addressed. These are beyond the scope of this paper.

In this paper, we will describe how a compiler can be constructed from the formal definition of a language. This compiler has the property of being correct by construction, and hence demonstrably correct. We will do this by defining a semantics for a small (but by no means trivial) high level language, (which for the purposes of this paper we will call Tosca – “not a Toy language, for Safety Critical Applications”), then constructing a compiler from them. Note that we are not proposing a new language, but are rather demonstrating how the formal semantics of a given language can be used in trusted compiler development.

Earlier work on compiler correctness includes [McP66, MiW72, Mor73, Coh79, Pol81]. Work on generating a compiler automatically from a denotational semantics definition of the language includes [Mos75, Pau81, Pau82, Wan84, Lee89] and more recent work on semantics-directed compiler generation includes [HoJ90].

2. Semantics

In order to write a correct compiler, it is necessary to have a formally defined semantics of the language. There are several ways of defining the semantics of programming languages, each appropriate for different purposes:

An axiomatic semantics defines a language by providing axioms and rules of inference for reasoning about programs, for example:

\[
\text{skip}; \langle \text{stmt} \rangle = \langle \text{stmt} \rangle = \langle \text{stmt} \rangle; \text{skip} \quad (1)
\]

It is appropriate for showing that two programs have the same meaning (useful, for example, when doing program transformations for the purpose of optimisations), but is rather too abstract for defining a compiler.